Information Coding Methods

Digital Image and Sound Processing

assoc. prof. dr. Kęstutis Jankauskas
kestutis.jankauskas@ktu.lt

KTU. Department of Multimedia Engineering.

Today in the Slides

- Run-length (RLE) Coding
- Huffman Coding
- Lempel-Ziv-Welch (LZW) Coding
- Arithmetic Coding
- Data Compression Ratio
- Coding Applications

Digital Information

- Any form of a digital signal is rendered to a stream of bits
- Sequences of bits represent bytes, symbols, numbers, words…
- General coding algorithms operate bit sequences regardless of how provided information can be interpreted in audio or image processing

Digital Information Coding

- Coding is used to **compress** information or make information unreadable by changing its form
- **Encoding** converts information to a coded form and **decoding** converts it back to the original form
- Types of audio and visual information coding:
  - **Lossless** – all information is encoded
  - **Lossy** – only important information is encoded

[1]

Run-length Encoding (RLE)

- Stores **run length** and data **value** instead of repeating the same value many times
- **Run** is a sequence in which the same value is repeated several times
- Effective if **long runs** are present in data stream
- Example:
  - *Original data: TTTTAAAGTTTT*
  - *Encoded data: 4T3A1G4T*
- Decoding repeats the value as many times as given in **run length**

[2]

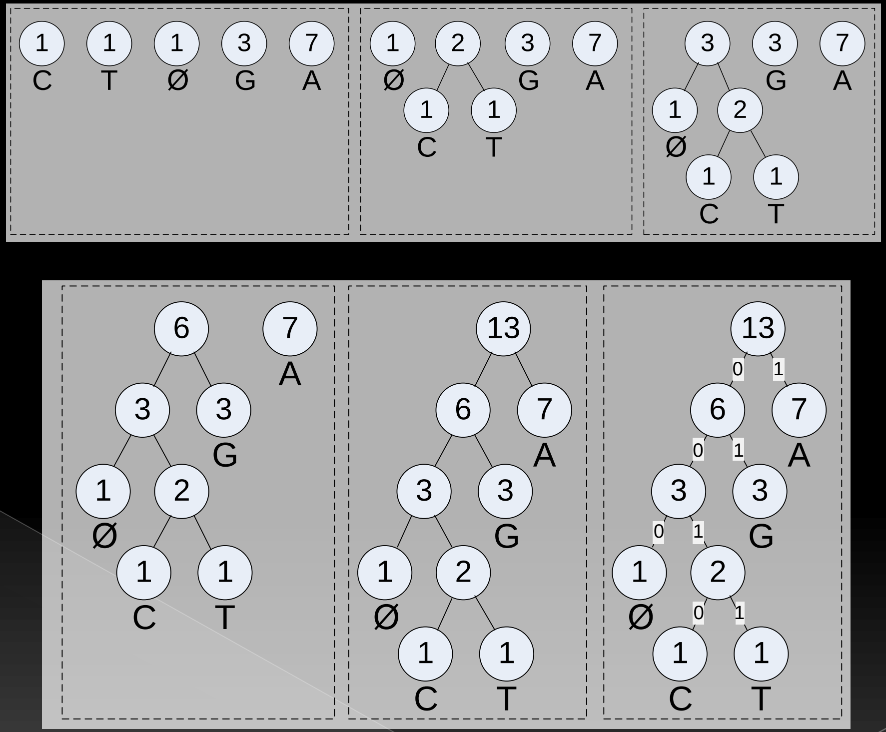
Huffman Encoding

- Gives each symbol (data block) a new **codeword**:
  - Frequent symbols are encoded with short codewords
  - Rare symbols are encoded with long codewords
- **Static Huffman encoding** is a three-step procedure:
  1. Counts symbol frequencies
  2. Constructs prefix code (binary **Huffman tree**)
  3. Encodes data
- It is possible to skip first two steps if the symbol occurrence is known and pre-generated or a standard Huffman tree is used

[3, 4]

Huffman Tree Construction

- Huffman tree construction is based on symbol frequency values:
  1. Tree node is constructed from two rarest symbols, its value is equal to the sum of child values;
  2. Tree node is placed in frequency list instead of child nodes, the list is sorted;
  3. 1 and 2 steps are repeated until only one element – Huffman tree – remains in the list.
- Tree leaf represents a symbol and path from a root to a symbol represents its new codeword
  - Left branch – 0, right branch – 1
  - Codeword has no identical match at the beginning of any other codeword



A – 1 G – 01 T – 0011
C – 0010 Ø – 000

[3, 4]

Huffman Decoding

- Original codewords (symbols) are recovered by parsing the compressed text with the coding tree
- The process begins at the root, it traverses left branch if “0” is read and right branch if “1” is read
- An original character is recovered when the tree leaf is reached
- The process is terminated when symbol “Ø” referring to the end of data is encountered

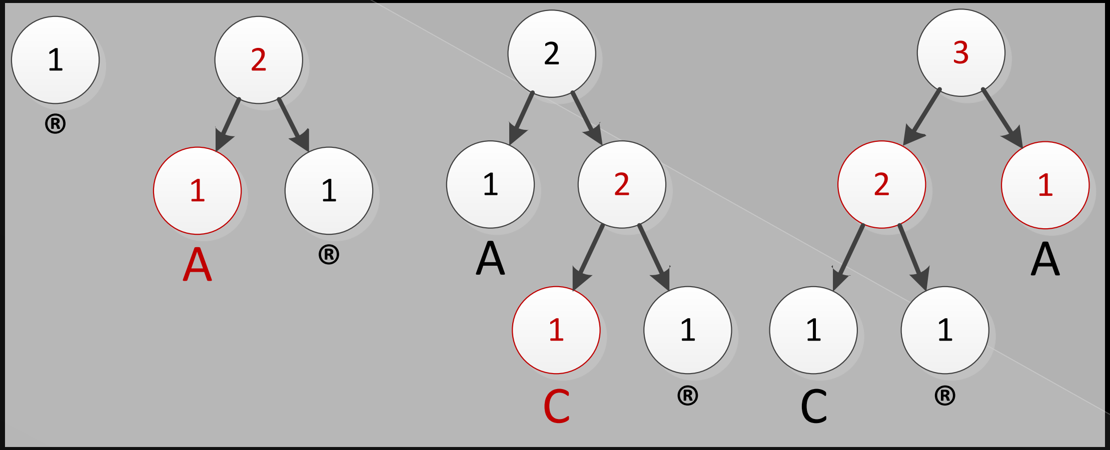
[3]

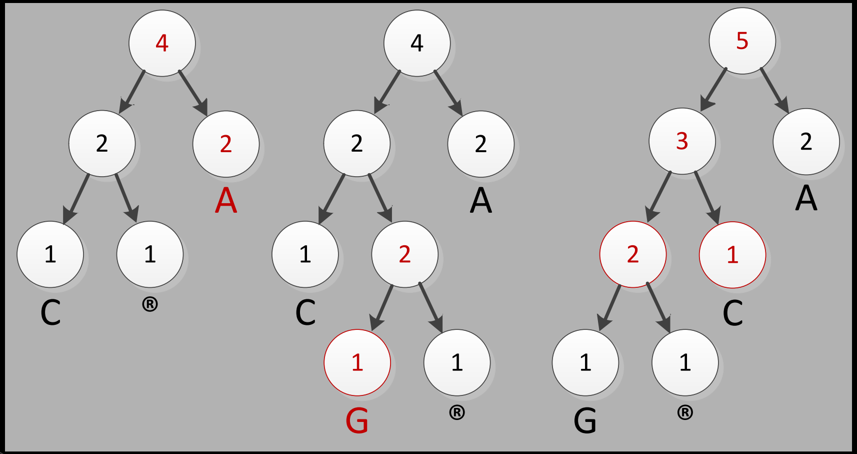
Dynamic Huffman Encoding

- Static Huffman encoding must **traverse data twice** and **include Huffman tree** into encoded data
- Dynamic (adaptive) Huffman encoding builds Huffman tree **during coding process**
- A tree is included into encoded data automatically
  - Artificial character “®”, which represents initial Huffman tree, is used
  - Nodes are sorted in descending (frequency) order

[3, 5]

Dynamic Huffman Encoding





Example

- Input 104 bits:

A	C	A	G	A	A	T	A	G	A	G	A	Ø
A code	C code	A code	G code	A code	A code	T code	A code	G code	A code	G code	A code	END code

- Output 61 bit:

A	®	C	A	®	G	A	A	®	T	A	G	A	G	A	®	Ø
A code	1	C code	1	01	G code	1	1	101	T code	0	100	0	100	0	111	END code

[3, 5]

LZW Encoding

- **Lempel-Ziv-Welch** (LZW) algorithm operates on the basis of the dictionary, which is build during coding process:
    1. Initial dictionary contains all possible strings of length one;
    2. Find the longest string *W* in the dictionary that matches the current input;
    3. Emit the dictionary index for *W* to output and remove *W* from the input;
    4. Add *W* followed by the following symbol in the input to the dictionary;
    5. Go to Step 2.

[6]

LZW Encoding

Example

- Input 125 bits
  - TOBEORNOTTOBEORTOBEORNOT#
- Output 96 bits
- Dictionary:

Symbol	Binary	Dec
Ø	00000	0
A	00001	1
…	…	…
Z	11010	26

Curr. Seq.	Next Char	Code	Bits	Extended Dictionary	
NULL	T				
T	O	20	10100	27:	TO
O	B	15	01111	28:	OB
B	E	2	00010	29:	BE
E	O	5	00101	30:	EO
O	R	15	01111	31:	OR
R	N	18	10010	32:	RN
N	O	14	001110	33:	NO
…	…	…	…	…	…
RN	O	32	100000	41:	RNO
OT	Ø	34	100010		
		0	000000		

[6]

LZW Decoding

- Decoding works by reading a value from the encoded data and outputting the corresponding string from the initialized dictionary
- Rebuilds the dictionary in the same way that an encoder does by concatenating symbols
- If variable-width codes are being used, the encoder and decoder must be careful to change the width at the same points in the encoded data

[6]

LZW Decoding

Example

- The same example from LZW encoding
- Decoder is always one step behind an encoder

Bits	Code	Output seq.	Extended Dictionary	
10100	20	T	27: T?	
01111	15	O	28: O?	27: TO
00010	2	B	29: B?	28: OB
00101	5	E	30: E?	29: BE
01111	15	O	31: O?	30: EO
10010	18	R	32: R?	31: OR
001110	14	N	33: N?	32: RN
001111	15	O	34: O?	33: NO
…	…	…	…	…
100000	32	RN	41: RN?	40: EOR
100010	34	OT	42: OT?	41: RNO
000000	0	∅		

[6]

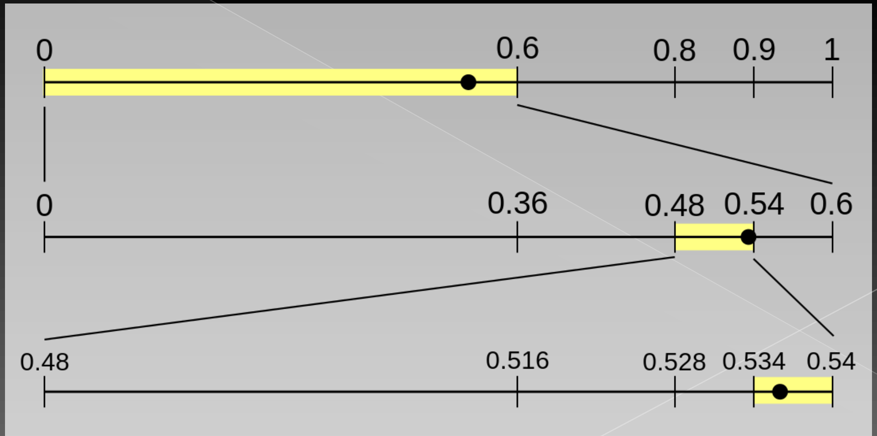
Arithmetic Encoding

- Frequently used symbols will be stored with less bits than rarely used symbols
- Is based on symbol occurrence probability and encodes the entire message into a single number
- Divides finite calculation range into intervals corresponding to probabilities, while an encoded message is represented by a point

Example

- Probabilities:

  A 60%, C 20%, G 10%, Ø 10%
- Input: AGØ
- Output: 0.538



[3, 7] [https://www.youtube.com/watch?v=FdMoL3PzmSA]

Binary Arithmetic Encoding

- The same idea can be implemented in binary form - Binary Arithmetic Coding (**BAC**)
- Enough bits must be used to obtain required precision
- Lower and higher interval boundaries are recalculated after each input symbol has been encoded:

$$\boxed{l = l + \frac{(h - l + 1) * F_i}{F_0}}$$

$$\boxed{h = l + \frac{(h - l + 1) * F_{i-1}}{F_0} - 1}$$

l – lower boundary

h – higher boundary

F_i– i-th symbol cumulative frequency

F_0 – total cumulative frequency

[3, 7]

Binary Arithmetic Encoding

Example

- Input: ACAGAATAGØ
- 8 bits are used ([0, 255) range)
  - [0, 255) -A-> [204, 255)
    - Out: 11, interval: [48, 255)
  - [48, 255) -C-> [96, 231)
    - Out: 10, interval: [96, 231)
  - [96, 231) -A-> [193, 231)
    - Out: 11, interval: [4, 159)
  - …

		A	C	G	T	Ø
i	0	1	2	3	4	5
F	5	4	3	2	1	0
f	0	1	1	1	1	1

		A	C	G	T	Ø
i	0	1	2	3	4	5
F	6	4	3	2	1	0
f	0	2	1	1	1	1

		A	C	G	T	Ø
i	0	1	2	3	4	5
F	7	5	3	2	1	0
f	0	2	2	1	1	1

f – frequency F– cumulative frequency

[3, 7]

Binary Arithmetic Decoding

- Decoding is performed in reverse to encoding process
- Window size (bit count of currently analyzed value x) is the same as for encoding
- Decoded symbol is the first symbol that:

$$\boxed{F_i > \frac{(x - l + 1) * F_0 - 1}{h - l + 1}}$$

l – lower boundary

h – higher boundary

x – analyzed value

F_i – i-th symbol cumulative frequency

F_0 – total cumulative frequency

- l and h are updated in the same way as in an encoder

[3, 7]

Binary Arithmetic Decoding

Example

- Input: 11101100111011111010000…
  - Value 236 = 11101100, F = 4
  - [0, 255) -A-> [204, 255)
    - out: A, shift: 2, interval: [48, 255)
  - Value 179 = 10110011, F = 3
  - [48, 255) -C-> [152, 185)
    - Out: C, shift: 2, interval: [96, 231)
  - Value 206 = 11001110, F = 5
  - [96, 231) -A-> [193, 231)
    - Out: A, shift: 2, interval: [4, 159)
  - …

[3, 7]

		A	C	G	T	Ø
i	0	1	2	3	4	5
F	5	4	3	2	1	0
f	0	1	1	1	1	1

		A	C	G	T	Ø
i	0	1	2	3	4	5
F	6	4	3	2	1	0
f	0	2	1	1	1	1

		A	C	G	T	Ø
i	0	1	2	3	4	5
F	7	5	3	2	1	0
f	0	2	2	1	1	1

Data Compression Ratio

- The efficiency of compression is defined by **data compression ratio**
- Lossless compression preserves all the information, but, in general, it does not achieve compression ratio much better than 2:1
- Compression algorithms which provide higher ratios either incur very large overheads or work only for specific data
- Lossy compression can achieve much higher compression ratios by removing information
- Compression ratio of at least 50:1 is needed to get 1080i video into a 20 Mbit/s MPEG transport stream

$$\boxed{CR = \frac{Uncompressed\ size}{Compressed\ size}}$$

[8]

General Coding Algorithm Applications

- **RLE** is used in: GIF, JPEG
- **Huffman coding** is used in: JPEG, MP3
- **LZW coding** is applied in: GIF, TIFF, PDF
- **Arithmetic coding** is used in: Context-adaptive BAC in MPEG AVC and HEVC

[2, 4, 6, 9]

Interesting Facts

- LZW has many variants like LZ77, LZ78, LZMA, LZSS, LZJB, etc., which include various modifications and adaptations for specific data sets
- DEFLATE is an efficient synthesis of LZW and Huffman algorithm that is used in PNG and ZIP formats
- Arithmetic coding is the generalization of Huffman coding and can often achieve better compress ratios
- AC and Huffman are considered entropy encoding compression schemes that are independent of the specific characteristics of the medium

[6, 4, 7, 10, 11]

References

1. https://en.wikipedia.org/wiki/Code
2. https://en.wikipedia.org/wiki/Run-length_encoding
3. Crochemore, M.; Lecroq, T. Text Data Compression Algorithms. from the book Atallah, M., J.; Blanton, M. Algorithms and Theory of Computation Handbook: General Concepts and Techniques, 2010
4. https://en.wikipedia.org/wiki/Huffman_coding
5. https://en.wikipedia.org/wiki/Adaptive_Huffman_coding
6. https://en.wikipedia.org/wiki/Lempel%E2%80%93Ziv%E2%80%93Welch
7. https://en.wikipedia.org/wiki/Arithmetic_coding
8. https://en.wikipedia.org/wiki/Data_compression_ratio
9. https://en.wikipedia.org/wiki/Context-adaptive_binary_arithmetic_coding
10. https://en.wikipedia.org/wiki/DEFLATE
11. https://en.wikipedia.org/wiki/Entropy_encoding